VM: The Saga Continues

- Topics
  - Where are we?
  - When memory needs exceed capacity: paging
  - Paging: who to evict
  - Working sets
- Learning Objectives:
  - Identify strategies for efficiently sharing physical memory.
  - Define a page fault and explain how they occur and are handled.
  - Explain the MIN, LRU, Clock, and Working set paging algorithms.
  - Tackle Assignment 3.

Where are we?

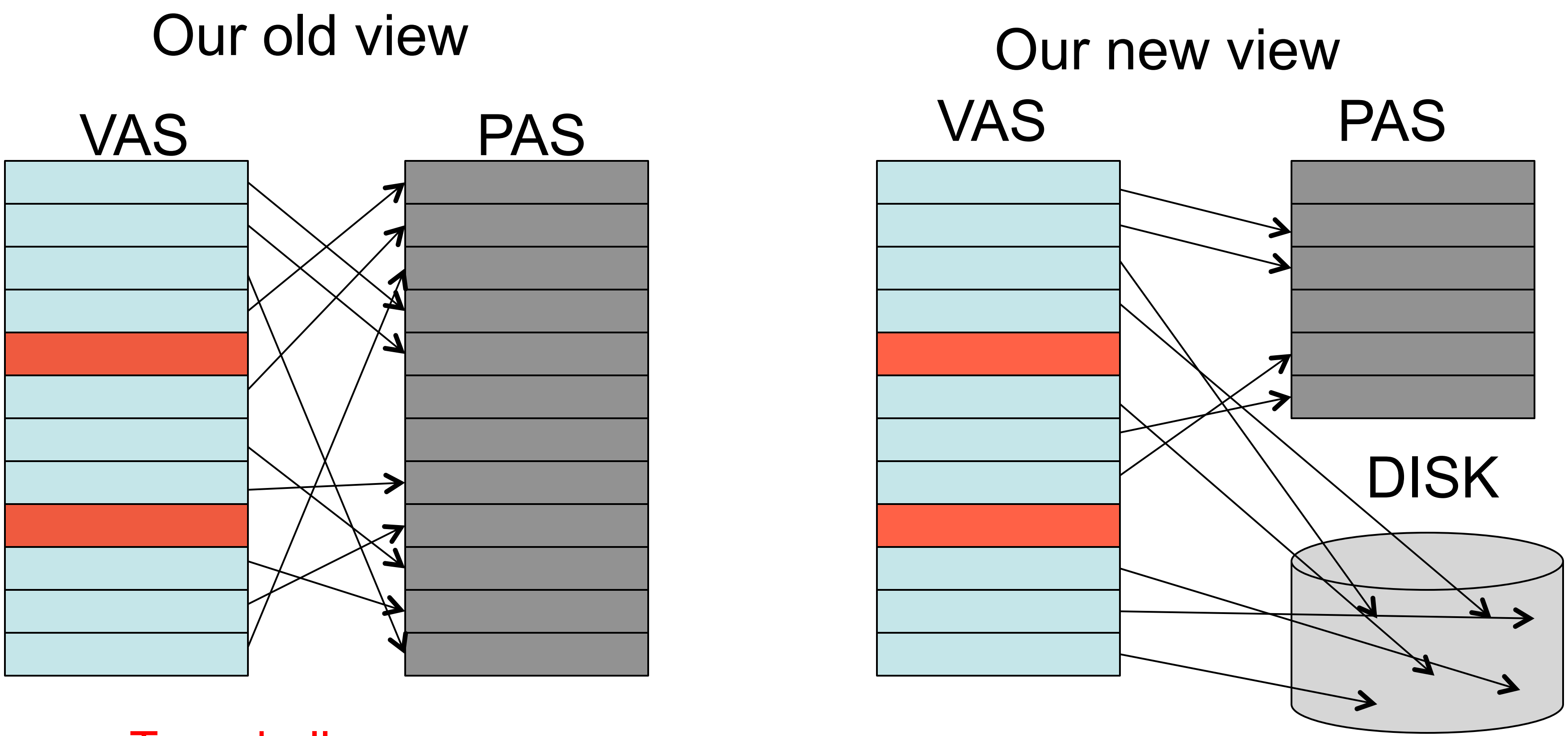
- Virtual Memory so far:

- What problem haven't we solved?

What is Paging?

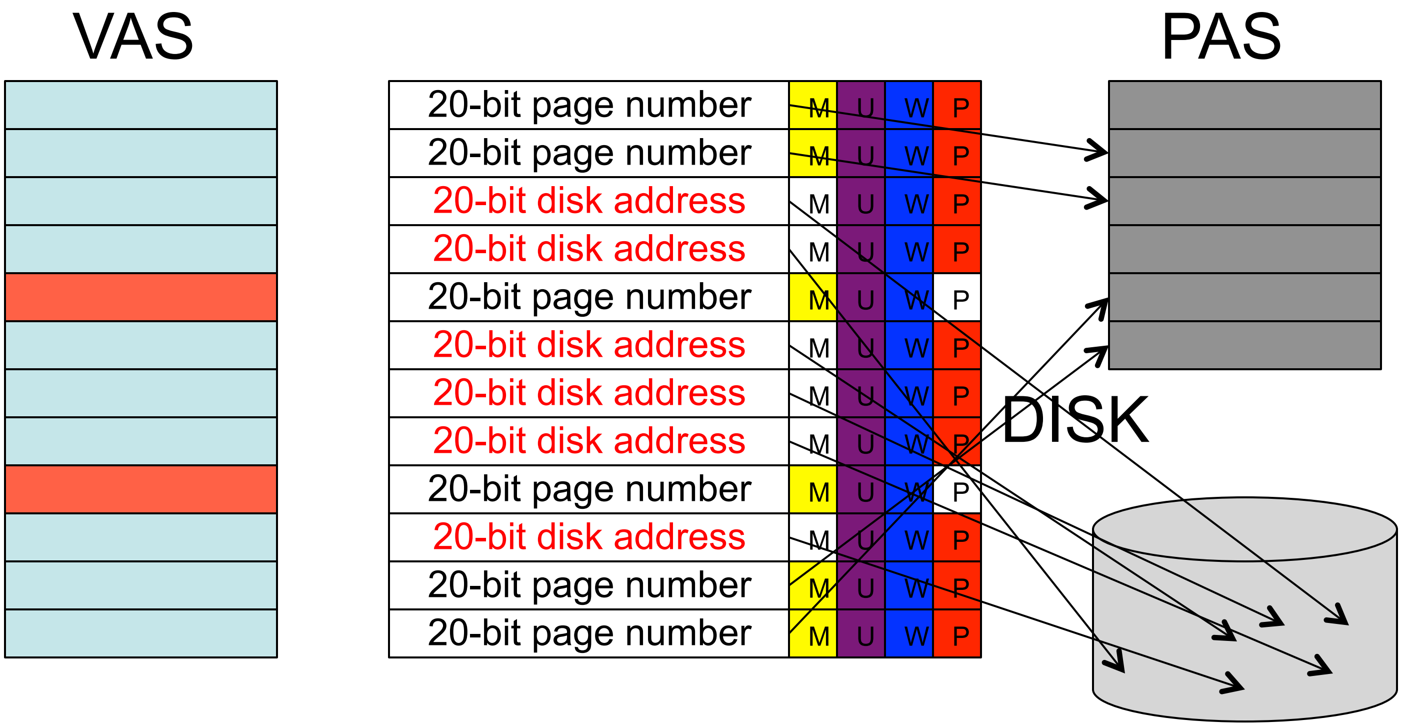
- The mechanism by which we allow processes to run with only some of their pages resident in memory.
- In a demand paging system, virtual pages can be in one of three states:
  - Memory resident: everything we've talked about so far.
  - Unmapped: there is nothing present at a virtual address.
  - Disk resident: there exists something at this VA, but it's not currently in memory.
- Pages in main memory are frequently called page frames.
- Pages on disk are frequently called backing frames.
- Our goal is to provide the illusion that main memory is as large as disk and as fast as memory.
  - When things go wrong, you get the feeling that memory is as small as memory and as slow as disk!
  - Fortunately, locality saves us (in most cases).

Our New View of Memory



- Two challenges:
  - How to run processes with some pages are missing
  - How to schedule which page are in main memory?

Extending PTEs

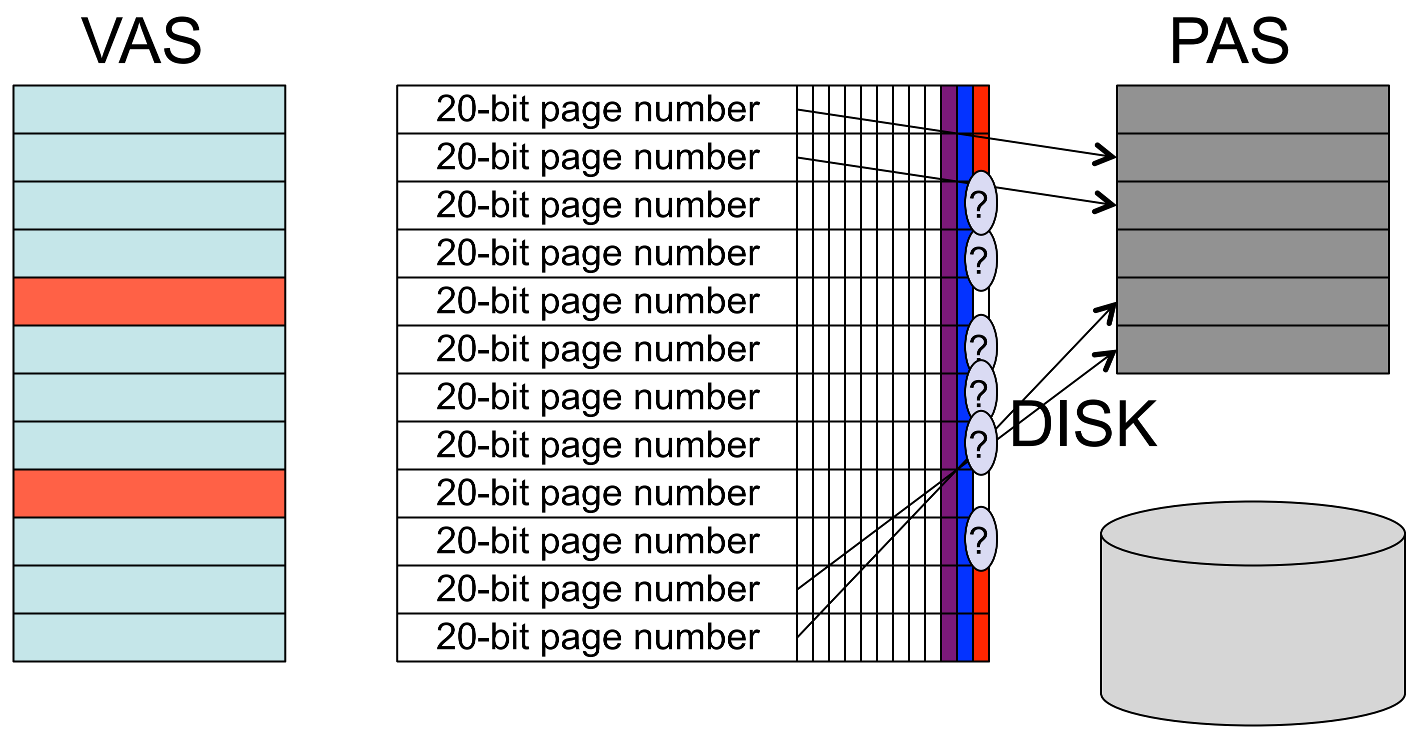


Let's add an "in-memory" bit that indicates if the page Is in-memory; when 0, the page has been swapped out.

Page Faults

- Extend page table entry (PTE) to include a bit that indicates if the page is in-memory.
- If virtual to physical translation yields a page table entry in which this bit is not set, the reference results in a trap, called a page fault.
- Any page not in main memory has an in-memory bit of 0.
- When a page fault occurs:
  - Operating system brings page into memory.
  - Page table is updated; in-memory bit is set.
  - Update TLB*
  - The process that faulted continues execution.
- Continuing a process is extremely tricky.
  - Page fault may have occurred in the middle of an instruction.
  - Need to make the fault invisible to the user process.

What do we do on the x86?



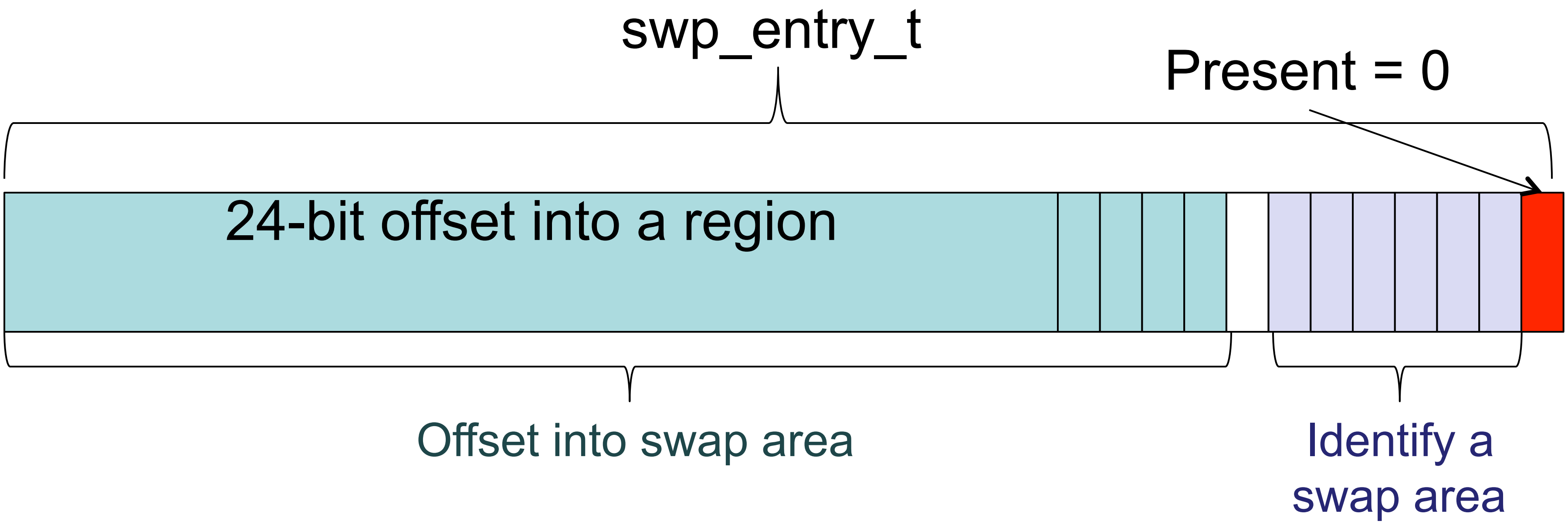
The x86 does not have an in-memory bit!

Translations are in hardware; if the page is not in-memory, then the hardware cannot translate it. What do you do???

Exercise 1

- On the x86, the operating system gains control any time a page in the VAS is not in-memory (even if the memory access is to a valid virtual address).
- Think about what information you need to store in the PTE to let you find a page that you have stashed away on disk.
- Design:
  - A PTE that describes an on-disk page (how can you tell the difference between an on-disk page and a page that is invalid in the VAS?).
  - Data structures to describe what is stored on disk.

Linux Paging (1)



If a swap entry references a 4 KB page, what is the maximum size of a swap area?

Linux Paging (2)

Maintain an array of structures, each of which describes a swap region:

```
struct swap_info_struct{
    unsigned int flags;        /* Indicates if entry is inuse or not. */
    struct file *swap_file;    /* Where the swap data lives on-disk */
    unsigned char *swap_map;   /* For each swapped-out page, stores a
                                * reference count of how many tasks
                                * use that page */
    unsigned int max;          /* Number of entries in swap_map */
    unsigned int inuse_pages;  /* Number of swap entries that currently
                                * contain a virtual memory page */
    unsigned int lowest_bit;   /* First possible free slot in swap_map */
    spinlock_t lock;
    …                          /* And other fields ... */
};
struct swap_info_struct *swap_info[MAX_SWAPFILES];
```

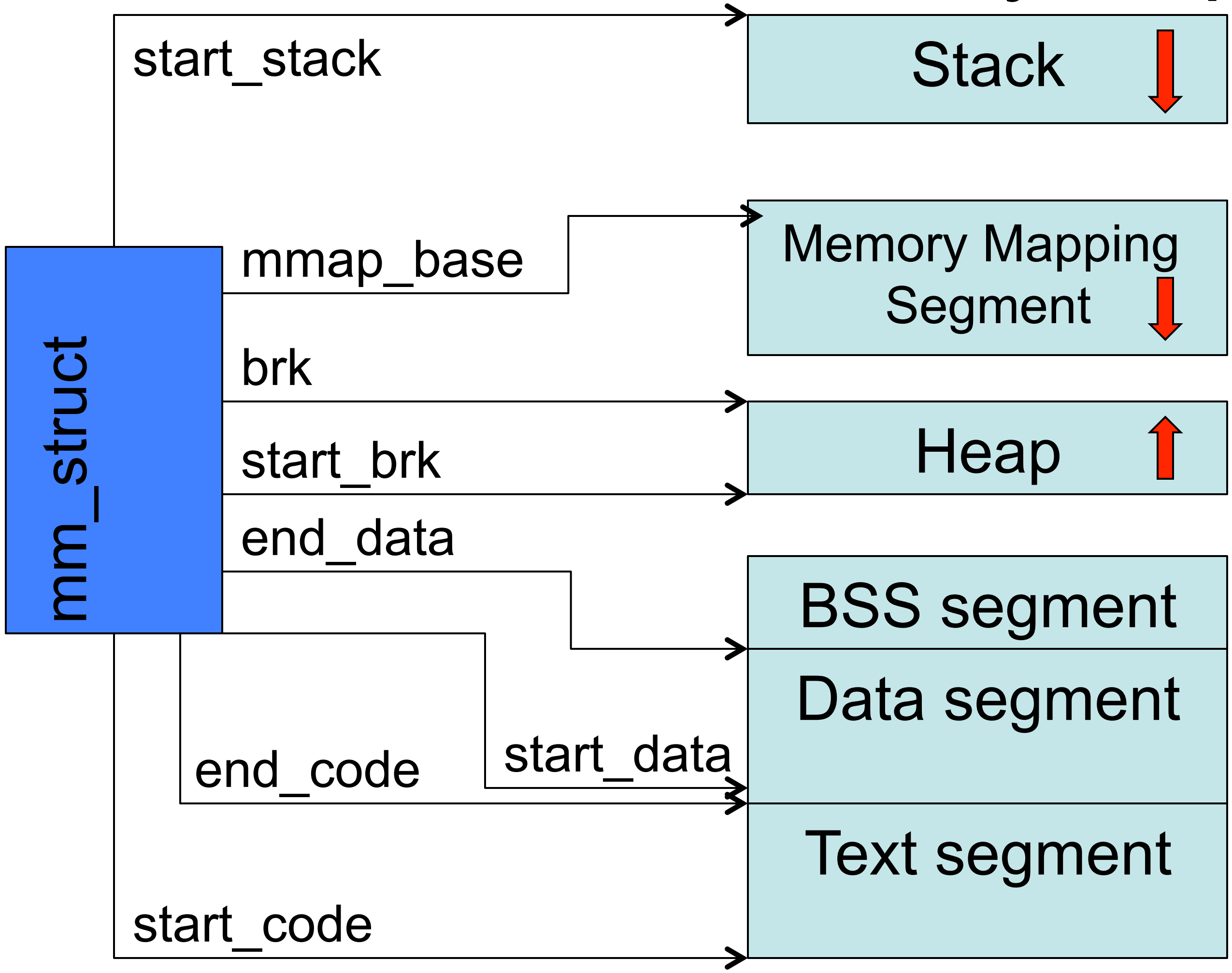
Linux Address Space Management

- Linux uses the task_struct to represent a process.
- Inside the task_struct, you'll find an mm_struct.
- The mm_struct is a summary of a process's virtual address space, containing:

  struct vm_aera_struct *mmap;

  ```
  unsigned long start_code, end_code;
  unsigned long start_data, end_data;
  unsigned long start_brk, brk;
  unsigned long start_stack;
  ```
- (as well as a ton of other stuff)

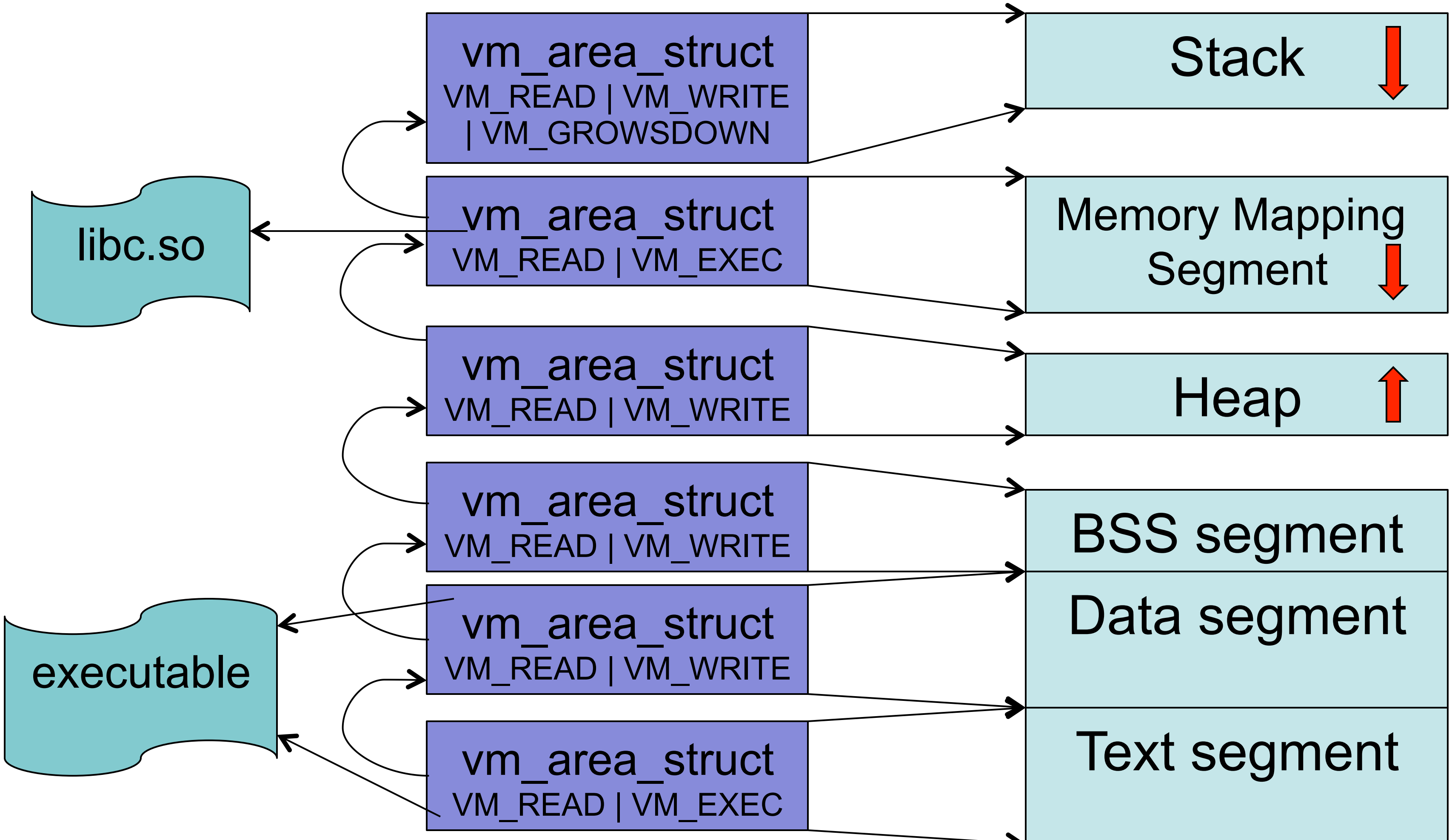
Parts of a Linux Memory Map (1)



Parts of a Linux Memory Map (2)

- Linux describes each of these parts of the VAS using a virtual memory area (VMA).
- A VMA describes a contiguous chunk of the VAS.
- Each VMA is described by a vm_area_struct, which contains (among other things):
  - Start and end address of the region
  - Pointer to its address space
  - Protection information
  - Links (to connect all the areas)
  - Information about the source of the area (e.g., file mapped)

Parts of a Linux Memory Map (3)



Exercise 2

- At this point, we've introduced examples of data structures that:
  - Facilitate hardware translation (TLBs and Page Tables)
  - Facilitate handling page faults (VMAs, Page Tables)
- What other algorithms and/or data structures might we need?
  1. Let's say that you have to bring a page in from swap; how do you decide where to place it in physical memory?
     - Design a data structure to handle this case.
  2. Let's say that memory is full and you need to kick out a page, how do you decide what page to kick out (evict?)
     - Think about what goals you want to achieve
     - Propose an algorithm or two to accomplish your goal

!!!Copy-on-write Pages

- !!!Useful for fork()
  - !!!OS initially marks pages as read-only
  - !!!On page fault caused by write, the OS gives each process its own version of the page
  - !!!Make a reference back to the MOD page fault on MIPS (which indicates that a process tried to write a page that doesn’t have the writable bit set)

More data structures: Core Map

- Core map maps physical addresses to virtual addresses.
- Uses of core map:
  - Find a free spot (page frame) into which a new page can be allocated.
  - Pre-emptively write dirty pages to disk.
  - Record space consumed by the operating system (so you don’t inadvertently allocate that space to user processes!)

!!!Huge Pages

- !!!Define TLB reach

Page Fault Handling Mechanics (1)

- Typically, the PC is incremented at the beginning of the instruction cycle. Therefore, if you do not do anything special, you will continue running the process at the instruction after the faulting one and it will appear as if the faulting instruction got skipped.
  - Users probably will not like this behavior.
  - "Hi, we're giving you virtual memory. Oh by the way, sometimes we skip instructions."
- You have three options:
  - Restart the instruction: undo whatever the instruction may have already done and then reissue the instruction.
    - Used by PDP-11, **MIPS R3000**, and most modern architectures.
  - Complete the instruction: continue where you left off.
    - Used in the Intel x86.
  - Test for faults before issuing the instruction.
    - Used in the IBM 370.

Page Fault Handling Mechanics (2)

- Without hardware support, you should either forget about paging or use complex (and disgusting) solutions.
  - MC68000, Intel 8086 and 80286: could not restart instructions.
  - Apollo systems (used Motorola CPUs) had two CPUs.
    - One executed user code.
    - If it took a fault, the user CPU stalled while the OS CPU fetched the page.
    - Once it got the page, the user CPU was un-stalled.
- Even with hardware support, the page fault handler must be able to recover the cause of the fault and enough of the machine state to continue the program.

Algorithm: Page Replacement

- If all our processes fit comfortably in memory, life is good.
- Life is rarely good!
- Page replacement is the act of selecting a page in memory for eviction.
- Selecting such pages badly can have dire performance consequences!

Page Replacement

- Random
  - Pick any page to evict.
  - Works surprisingly well!
- FIFO
  - Throw out page that has been in memory the longest.
  - The basic idea is that you give all pages equal residency.
- MIN
  - Predict the future.
  - Evict the page that will not be referenced for the longest time.
  - Tough to implement.
  - Good for comparison.
  - Defined by Laszlo Belady (known as Belady's algorithm).
- LRU
  - As usual, use past to predict future.
  - Evict page that has been unreferenced the longest.
  - With locality, this is a good approximation to MIN.
- What makes implementing some of these difficult? What other metrics/ statistics might you want to keep about your pages?

Page Replacement

- Random
  - Pick any page to evict.
  - Works surprisingly well!
- FIFO
  - Throw out page that has been in memory the longest.
  - The basic idea is that you give all pages equal residency.
- MIN
  - Predict the future.
  - Evict the page that will not be referenced for the longest time.
  - Tough to implement.
  - Good for comparison.
  - Defined by Laszlo Belady (known as Belady's algorithm).
- LRU
  - As usual, use past to predict future.
  - Evict page that has been unreferenced the longest.
  - With locality, this is a good approximation to MIN.
- What makes implementing some of these difficult? What other metrics/statistics might you want to keep about your pages?
  - LRU is recency; requires a single queue
  - Frequency is easier (sorting is hard).

Playing pager (3 memory frames)

Reference stream	**A**	**B**	**C**	**A**	**B**	**D**	**A**	**D**	**B**	**C**	**B**
FIFO	A										
		B									
			C								
MIN	A										
		B									
			C								
LRU	A										
		B									
			C								

Playing pager (3 memory frames)

Reference stream	A	B	C	A	B	D	A	D	B	C	B
FIFO	A					D				C	
		B					A				
			C						B		
MIN	A									C	
		B									
			C			D					
LRU	A									C	
		B									
			C			D					

- Just like STCF, MIN is optimal, but not implementable.
- Just like priority queues or fair-share scheduling, use the past to predict the future. For page replacement, LRU (least recently- used) works remarkably well.

Implementing LRU

- Need hardware to keep track of recently used pages.
- Perfect LRU?
  - Register for every physical page.
  - Store clock on every access.
  - To replace, scan through all the registers.
  - Assessment?
    -
    -
- Approximate LRU
  - Find any *old* page.
  - May not be oldest, but if it's old, it's probably good enough.
  - After all, LRU is an approximation of MIN; what's another level of approximation?
- Clock
  - Maintain a *use* bit for each frame.
  - Set bit on every reference.
  - Operating system sweeps through memory clearing use bits.

Implementing LRU

- Need hardware to keep track of recently used pages.
- Perfect LRU?
  - Register for every physical page.
  - Store clock on every access.
  - To replace, scan through all the registers.
  - Assessment?
    - Expensive!
    - Not very practical.
- Approximate LRU
  - Find any old page.
  - May not be oldest, but if it's old, it's probably good enough.
  - After all, LRU is an approximation of MIN; what's another level of approximation?
- Clock
  - Maintain a use bit for each frame.
  - Set bit on every reference.
  - Operating system sweeps through memory clearing use bits.

Implementing Clock

- When time to replace, replace a page frame with a 0 use bit.
- On page fault — circle around clock.
  - If bit is set, clear it.
  - If bit is not set, replace it.
  - Can this loop infinitely?
  - Can also incorporate *dirty* bit since dirty pages are more expensive to evict than clean ones.
- In clock, what does it mean if the clock hand is sweeping very slowly?
- What if the hand is sweeping very quickly?

Implementing Clock

- When time to replace, replace a page frame with a 0 use bit.
- On page fault — circle around clock.
  - If bit is set, clear it.
  - If bit is not set, replace it.
  - Can this loop infinitely? NO
  - Can also incorporate *dirty* bit since dirty pages are more expensive to evict than clean ones.
- In clock, what does it mean if the clock hand is sweeping very slowly?
  - Plenty of memory.
  - Not many page faults.
  - This is good (desirable).
- What if the hand is sweeping very quickly?
  - Not enough memory.
  - Thrashing.